Autotasked Performance in the NAS Workload: A Statistical Analysis

R.L. Carter[1] and *I.E. Stockdale*[2]

RND-92-021 December 1992

NAS Systems Development Branch
NAS Systems Division
NASA Ames Research Center
Mail Stop 258-6
Moffett Field, CA 94035-1000

[1]NASA Ames Research Center, Moffett Field, CA 94035-1000
[2]Computer Sciences Corporation, NASA Ames Research Center, Moffett Field, CA 94035-1000

Autotasked Performance in the NAS Workload: A Statistical Analysis RND-92-021

R.L. Carter[1] and *I.E. Stockdale*[2]
NASA Ames Research Center
Moffett Field, CA 94035, USA

Abstract

A statistical analysis of the workload performance of a production quality FORTRAN code for five different Cray Y-MP hardware and system software configurations is performed. The analysis was based on an experimental procedure that was designed to minimize correlations between the number of requested CPUs and the time of day the runs were initiated. Observed autotasking overheads were significantly larger for the set of jobs that requested the maximum number of CPUs. Speedups for UNICOS 6 releases show consistent wall clock speedups in the workload of around 2, which is quite good. The observed speedups were very similar for the set of jobs that requested 8 CPUs and the set that requested 4 CPUs.

The original NAS algorithm for determining charges to the user discourages autotasking in the workload. A new charging algorithm to be applied to jobs run in the NQS multitasking queues also discourages NAS users from using autotasking. The new algorithm favors jobs requesting 8 CPUs over those that request less, although the jobs requesting 8 CPUs experienced significantly higher overhead and presumably degraded system throughput.

A charging algorithm is presented that has the following desirable characteristics when applied to the data: higher overhead jobs requesting 8 CPUs are penalized when compared to moderate overhead jobs requesting 4 CPUs, thereby providing a charging incentive to NAS users to use autotasking in a manner that provides them with significantly improved turnaround while also maintaining system throughput.

---

[1]The author is employed in the NAS Systems Division at NASA Ames Research Center.
[2]This work was supported by NASA Contract No. NAS2-12961 while the author was an employee of Computer Sciences Corporation under contract to the Numerical Aerodynamic Simulation Systems Division at NASA Ames Research Center.

1. Introduction

There is a great deal of interest in improving the throughput of Computational Fluid Dynamics (CFD) codes on multi-processor vector supercomputers by allowing the code to take advantage of more than one processor at a time. Traditional shared memory multiprocessor supercomputers such as those produced by Cray Research and Convex, support either manual (multi-tasking) or automatic (automatic parallelization) generation of parallel FORTRAN. Cray Research (CRI) automatic parallelization (hereinafter referred to as autotasking) also allows for additional manual intervention. System administrators feel that such codes may better utilize system main memory. In the absence of parallel codes, some processors may be idled because all available memory is in use. This report describes an experiment in which a sample NASA CFD code is used to estimate the *practical* benefits obtained by users and system managers due to running autotasked codes in the NAS workload.

1.1 Parallel Performance Data and Production Environments

There is a pressing need for performance data on parallel codes in production environments in order to manage the system efficiently and to advise users how best to run their application codes in a production workload. Obtaining quantitative estimates of the performance of parallel codes in production workloads presents substantial difficulties. Components of the workload change constantly and nonlinearly over time, and the operating system and hardware configurations may vary significantly. Thus there is relatively little published data about the performance of parallel codes in real production environments. Lack of data has led to simplified, indirect approaches to assessing the workload performance of particular codes. Two of the most common methods of making such assessments are through the use of dedicated time studies and by observing the performance of carefully constructed synthetic workloads.

Studies of the suitability of particular codes for effective parallelization initially focus on the performance of the parallel version relative to the unitasked version in a *dedicated* environment [1,2,3,4,5]. Performance may be substantially lower in the production workload than in a dedicated environment. This is reasonable, as measurements of execution times in such a workload may vary widely due to factors unrelated to the particulars of the code, such as system scheduling parameters, and system load. However, neglecting these factors limits the capability of such studies to predict workload performance of parallelized production codes.

A more direct approach measures the performance of parallel codes in *synthetic* workload environments [6,7,8,9,10,11]. These experiments typically consist of a suite of codes in which particular parallel applications compete for processor (CPU) time under actual scheduling constraints. The experiments are run in dedicated time, with the usual goal of observing overall synthetic

workload throughput. The operating system environments may be tuned to obtain some range of results as an estimate of the systematic error present in trying to model a variety of (or even one) real-life workloads. While this approach is superior to using dedicated time single code results, it remains an approximation to what occurs in a heavily loaded production environment.

Finally, measurements of parallel code performance have been made in production workload environments [12]. There is little detailed analysis of the connection between system parameters and observed elapsed times, however. These measurements also lack statistical precision. At most, three to four runs of a particular code are used to estimate its performance in the workload. These experiments use the NAS Kernels program, rather than a production code. While a kernel may simulate the workload as a whole, the typical NAS supercomputer user will be running a production CFD code rather than simple kernels. In the following, a reference to a *workload* always implies the NAS production workload unless explicitly stated otherwise.

The existing data constitute a number of very interesting pieces to a jigsaw puzzle. The simulated workloads provide a controlled environment in which to assess the importance of various parameters and conditions. The production results provide a glimpse of actual code performance in workloads. Nevertheless, essential information is not yet available. There is no information on the sensitivity of job turnaround time to system parameters and system load. The existing production workload data is based on a small number of runs of each job. Given the large fluctuations in execution times, it is difficult to estimate the benefit which a user may actually expect to see by parallelizing a code. By the same token, it is difficult for a system administrator to set charging policies which reward multitasking or autotasking in accordance with the need for high overall throughput.

1.2 NAS User Community

The NAS user community consists of over 1100 researchers at more than 150 sites across the nation. NAS users in 1990 consumed over 114,000 Cray hours while working on more than 400 scientific projects. Most users are working on the types of aeroscience problems that NAS was created to address. These problems include direct solutions of the Navier-Stokes or Euler equations. Others are pursuing applications which range from modeling the structure and properties of superconducting compounds, to studying the fluid flow inside an artificial heart, or investigating the structure of the AIDS virus. These users are typically conventional FORTRAN programmers, using single processor versions of vectorized codes.

1.3 Cray Research Inc. Y-MP System

The Cray Y-MP is an eight CPU, multiple instruction multiple data stream (MIMD) computer system. The clock period (CP) at the inception of the project was 6.33 nanoseconds. Shortly after the inception of this project the clock speed was reduced to 6.0 nanoseconds, and the main memory was

increased from 32 Megawords (MW) to 128 (MW). The relatively fast shared main memory is augmented by a slower 256 MW solid-state storage device (SSD).

The operating system (UNICOS) is UNIX System V based, and underwent over five significant upgrades over the course of this study. Support exists for multiprogramming, where different processors run different jobs, and *multitasking*. Multitasking allows multiple processors to execute two or more parts of a single program in parallel [1]. The effects of multitasking may be obtained by invoking the autotasking options of the FORTRAN compiler. FORTRAN codes are by default run with full vectorization on a single processor, or *unitasked*. A *job* is a single program that may or may not spawn multiple processes. Jobs may be run in batch or dedicated mode. Batch mode jobs run in a multi-programming environment concurrently with other batch jobs and interactive users. Scheduling of batch jobs is handled by the Network Queuing System (NQS) [13].

Dedicated mode jobs run interactively with minimum competitive system and user activity. The performance of a code running in a multiprogramming, multiprocessing environment strongly depends on the availability of system resources such as processors, disks, or memory. Programs run in dedicated mode have minimal competition for resources and hence obtain the best performance possible on the Cray Y-MP. This performance is also much more reproducible than that measured in the workload environment. Jobs run in batch mode on the Y-MP under normal system workloads compete for resources with other jobs and are subject to being swapped to disk. This adversely affects average wall clock execution time performance compared to programs run in dedicated time.

1.4 Performance Evaluation Tools.

There are a number of tools which facilitate the parallelization of, and evaluate the performance of, a FORTRAN program on the Y-MP. A short description of the tools used in obtaining and analyzing the data is provided below. Tools provided by Cray Research Inc. are described in more detail in [15].

The Y-MP hardware performance monitor *hpm* summarizes the machine performance of a program by reporting various hardware counter statistics. Statistics are arranged by groups. Each group reports a set of related statistics on aspects of the program's performance. Group 0 is an execution summary which reports such statistics as instructions issued, I/O and CPU references, and floating point additions, multiplications, and reciprocals. Group 1 reports on various hold issue conditions. Group 2 summarizes memory activity, while Group 3 reports vector events and an instruction summary. Preparation involves normal compilation of the program source which is then run with the *hpm* command on the command line in the same

manner as the UNIX *time* command. *hpm* works with multitasked programs.

The *ja* command is a UNICOS utility that provides accounting information on program runs. Useful statistics include elapsed time, user CPU time, system CPU time, concurrent CPUs, and average concurrent CPUs.

The *nasja* command is a local utility that provides additional accounting information. Charging incentives for billing multitasked jobs are visible as negative billing charges for time accumulated on two or more concurrent CPUs.

The *schedv* command is a UNICOS utility that sets and reports memory scheduling parameters. These parameters include criteria for determining whether or not a job is a CPU or memory hog, and thus subject to swapping by the scheduler. They also include parameters which are used in determining swap-in/out priorities of jobs in the workload.

The *sar* command is a UNICOS utility that provides comprehensive information on operating system activity. Information such as portion of time in user and system modes, portion of time in user, UNIX and idle for each processor, buffer activity, system swapping and memory moves are available for sampling time periods of user specified length and frequency.

The *ldave* command is a local utility that provides the number of interactive users and one minute, five minute, and fifteen minute floating averages of the overall system load. The system load is measured as the number of processes, both interactive and batch, waiting in the run queue.

The *mu* command provides in-core, swapped, and total memory, in Megawords. *Ldave* and *mu* are useful in obtaining a general measure of overall system load but unfortunately are too coarse grained to provide meaningful information for individual programs running in a normally loaded system.

The UNICOS *ps* command provides information on the status of running jobs, including status flags which indicate whether a job is swapped or running.

1.5 NAS Operations

For system configuration purposes, the accounting week is divided into prime time (0500—1800 Monday through Friday) and non-prime time (1800—0500 Monday through Friday and all day weekends). Prime time is considered to be the period when there is high interactive use of the system. System parameters were, for some of the OS configurations studied here, varied between prime and non-prime time in order to achieve better performance for interactive and batch processes, respectively. The NAS production environment over the period tested (October 30, 1989—December 19, 1991) consisted of a mixed interactive and NQS batch workload.

The Y-MP production workload quantifiably changed over the course of this experiment. Table 1 shows the mean values of several statistics reported by UNICOS during the execution time of the autotasked job used in this experiment. The characteristics of this job are described below. The *mu* and *ldave* data are consistent with a higher number of small memory interactive jobs during prime time. The *sar* data shows significantly more idle cycles during non-prime time. This in some (but not all) cases might be attributable to small memory NQS queues draining [*14*], which suggests that the throughput of the NAS Y-MP system was memory limited. This situation has been cited [6, 10] as amenable to overall system throughput improvement by including multitasked jobs in the workload, *i.e.*, multitasked jobs are able to take advantage of CPU cycles that would otherwise be wasted. The over-subscription of memory increased steadily over the duration of this experiment (*cf.* Table 1) , making autotasking more alluring.

Table 1
Measured System Load Statistics

UNICOS Version	Total MWords (from *mu*)		5 Min. Load Avg. (from *ldave*)		% Tot. Cycles Idle (from *sar*)	
	Prime	Off-Prime	Prime	Off-Prime	Prime	Off-Prime
5.0	37.2 ± 0.2	33.2 ± 0.5	20.5 ± 0.7	15.3 ± 0.4	0.20 ± 0.08	4.3 ± 0.8
5.1.10	109.8 ± 2.4	110.2 ± 1.2	23.3 ± 0.6	14.5 ± 0.3	1.06 ± 0.35	13.03 ±1.15
6.0/6.0.5	123.1 ± 4.6	115.2 ± 3.3	19.5 ± 1.0	14.7 ± 0.6	10.51 ± 2.26	16.84 ± 2.31
6.0.12	131.2 ± 4.4	117.6 ± 2.9	21.0 ± 1.3	14.4 ± 0.5	4.09 ± 1.35	7.78 ± 1.40
6.1.4/6.1.5	146.6 ± 2.1	135.2 ± 1.6	25.1 ± 0.9	16.0 ± 0.3	3.08 ± 0.49	11.85 ± 1.08

1.6 Parallel Job Statistics

The UNICOS environment variable NCPUS occurs frequently in the following, and denotes the number of concurrent processors requested by a job at runtime. The most important of the statistics reported by UNICOS for analyzing the runtime performance of a parallel code are the time statistics produced by *ja* and *hpm*. Several interrelated time quantities are reported for each UNICOS job.

We define the following times in units of seconds. Connect time to i concurrent processors $T_{c,i}$ is the elapsed time i processors were concurrently attached to the job (either executing or idle) and is obtained from the *ja* summary report. Total connect time T_{ct} (also denoted total CPU time) is the the sum of the $T_{c,i}$ and is obtained from the *ja* summary report. The variable $\langle n_{CPUs} \rangle$ is the average number of concurrent processors as reported by *ja*:

$$\langle n_{CPUs} \rangle = \frac{\Sigma i T_{c,i}}{T_{ct}}$$

The notation $<n_{CPUs}>$ indicates the mean CPU time for all runs with NCPUS requested CPUs. The summation runs over i, from 1 to NCPUS. In the following, the bracket (<>) notation will be used for all arithmetic mean values; sigma (σ) will be used to represent the standard deviations.

Wait semaphore time T_{ws} is the sum of all the individual CPU execution times spent waiting for a semaphore (*i.e.*, waiting at a synchronization point). It is not directly relatable to execution time since concurrent processors waiting for a semaphore are simultaneously accumulating wait semaphore time. Wait semaphore time is obtained from the *hpm* group 1 output.

User CPU time T_{usr} is the difference between total connect time T_{ct} and the wait semaphore time T_{ws}.

$$T_{usr} = T_{ct} - T_{ws}$$

System CPU time T_{sys} is essentially UNIX system kernel work attributable to a job. System CPU time is obtained from the *ja* command and summary reports. Since the kernel is shared by all processes some work is not attributable to a unique process and relevant charges are distributed among all active processes. Similarly, some system work attributable to a job is not charged to the system CPU time.

The total CPU time T_{CPU} is defined to be:

$$T_{CPU} = T_{ct} + T_{sys}$$

This time (which includes the wait semaphore time) is used in calculating the autotasking overhead.

The turnaround time for a job is the elapsed time from the initial submittal of the NQS job to the completion of the job, and includes wallclock time as a component. Wallclock time is the elapsed time from the beginning of execution of the job to its completion.

Two statistics are used to evaluate the wallclock time of a job in the workload. The principal measure is the elapsed time reported by *ja*. This time, which is reported after the job has completed, is denoted $T_{wc,1}$. We also use the wallclock time printed out by the FORTRAN STOP command. This latter statistic, which we label $T_{wc,2}$, was available for all datasets except UNICOS 5.0.

The autotasking overhead $O_{A,n}$, where n is equal to NCPUS, is derived from the total CPU time, and is defined as:

$$O_{A,n} = \frac{<T_{CPU}>_n - <T_{CPU}>_1}{<T_{CPU}>_1}$$

The notation $<T_{CPU}>_n$ indicates the mean CPU time for all runs with n requested CPUs.

2 Design of this experiment

The measurement described in this report is an attempt to remedy some of the deficiencies present in previous results. A sample production code has been run in the workload over an extended period of time. The data collected gives results which have good statistical precision, and are not biased by the job mix present in the workload on any particular day or week. The UNICOS shell scripts which control the experiment have been instrumented with calls to the system performance evaluation tools described above. This instrumentation plays a critical role in assessing the impact of constantly changing scheduling parameters and system load on the code's performance.

The experiment has been run under a variety of system hardware and software configurations, giving the ability to assess each operating system and hardware improvement as it pertains to the performance of autotasked codes. Using the instrumentation referred to above, the importance of these upgrades may be compared to effects due to the system load and configuration.

2.1 Autotasked Code Description

The code chosen was SPARK, which is an implementation of a numerical model for supersonic reacting mixing layers as detailed in [16]. SPARK has been applied to problems associated with the propulsion systems of the National Aerospace Plane. The code used for this study solves the two-dimensional Navier-Stokes equations coupled to a two-species chemical reaction problem. The program was designed to consider the multicomponent diffusion and convection of important species in the chemical reactions, and the interactions between fluid mechanics, chemistry and thermodynamics. Specifically, the kinetics of the chemical reaction are incorporated into the simulation and in fact their computation constitutes the bulk of the computational expense.

The governing equations are discretized using a temporally implicit scheme which is solved by a modified MacCormack technique [16]. The FORTRAN source comprises approximately 5000 lines of code. Under UNICOS 5.0 the executable size of the unitasked job was 6.6 MW and the executable size of the autotasked job was 6.8 MW. Under UNICOS 5.0.12 the executable size of the unitasked job was 6.6 MW and the executable size of the autotasked job was 7.5 MW. Normal unitasked execution requires approximately 300 CPU seconds. Since the algorithm is time-stepping in nature, for the purposes of this study a shorter execution time of approximately 75 CPU seconds was chosen to allow reasonable turnaround time with the available NQS queues.

A complete description of the dedicated time performance of the code is given in [4]. The code was run in the NAS production environment with *fpp* options to enable inner loop autotasking and level 6 subroutine inlining

(cf77 -Zp -Wd,-ei6). Running with these options under UNICOS 5.0 with 32 MW of main memory and NCPUS set to 8, the code obtained 832.7 MFLOPS with a speedup of 5.0 in dedicated time. Using the same options under UNICOS 6.1.4, the code obtained 800.1 MFLOPS with a speedup of 4.1 (NCPUS set to 8) and 570.7 MFLOPS with a speedup of 2.9 (NCPUS set to 4) in dedicated time. These options were used for the data presented in this report. A more detailed history of the performance of the code over the UNICOS versions examined in this report is presented in [17].

2.2 Caveats

A number of compromises were made in order to obtain a realistic, consistent, and cost-effective measurement of the variation of wall clock turnaround time. The choice of a production code over a set of kernels or a synthetic benchmark represents the practice of a typical user. The drawback of this approach is that users' codes differ, and the behavior of SPARK may not be easily extrapolated to other codes.

The cost of the experiment was a consideration as well, and was designed to be minimized. The chosen code is regarded to be a typical example of a reasonably high performance NAS CFD code suitable for autotasking. There was significant doubt that increasing the experimental cost two or more fold (as would be required if a small number of additional codes were studied in the experiment) would generate a correspondingly significant improvement in experimental findings. The decision was made to restrict the number of codes studied to one, and the findings we present below must be viewed with this consideration in mind.

In order to have a constant measure of wallclock time in the NAS system, the memory requirements and execution time of SPARK were not changed. On the other hand, both the hardware and system software of the NAS Y-MP underwent significant upgrades. This means that, while initially SPARK required 1/4 of the Y-MP main memory, it required only about 1/16 of the 128 MW main memory after the upgrade. The NAS Y-MP has recently been upgraded to 256 MW; what was once proportionally a large memory job is now proportionally quite small. This is especially important to bear in mind when evaluating the effects of system parameters.

Finally, the job mix has changed as system management policies have changed. The NQS queue structure has undergone a number of changes. The load varies significantly during the operational year due to user demands stemming from conference deadlines, *etc.* These observations serve to underscore the point that the *only* parameter that can be considered to be constant over a series of hardware and software upgrades in the production system is the source code of the application software probe used in this investigation.

2.3 Methodology

The experiment was performed by running a group of shell scripts using the UNICOS *cron* facility. The startup script executed once every four hours, with the first run of the day occurring at 2 a.m. The startup script submitted a batch script to the NQS utility after waiting a random amount of time which varied uniformly between zero and four hours. After waiting in the NQS queue, the batch script ran the SPARK probe. The batch script was instrumented with calls to the system performance utilities described in Section 1.4 above. For the UNICOS 6.1 data only, the batch script spawned a background process which called *ps* periodically while SPARK executed.

Autotasked versions of SPARK were run with NCPUS equal to 1, 4, and 8, where NCPUS is a UNICOS environment variable specifying the number of CPUs requested by the job. The batch script randomly selects the value of NCPUS to be used for any given run. The UNICOS 6.1 dataset is an exception as part of the data was taken with NCPUS randomly chosen to be 4 or 8 only. For the UNICOS 5.0 and 6.1 datasets, a unitasked version of SPARK was run for comparison with the 1 CPU autotasked version. In this paper, we will refer to data taken with NCPUS set to n as "n CPU data," where n is 1, 4, or 8.

During periods of high system load, it was possible for the batch script to be in the NQS queue for more than four hours. When this occurred, the next batch job was not submitted. This resulted in a smaller statistical sample for the prime time runs than for the off-prime time runs.

The outputs of each run of the batch script were collected into a single "raw data" file. This file was then post-processed by an *awk* script in order to produce a "crunched" file containing one line of summary information for each run. The crunched file was imported into a Microsoft Excel spreadsheet on a Macintosh for detailed data analysis. Results obtained using Excel were independently checked with a data analysis program written in C on a workstation.

3 Results

3.1 Wall clock times for the tested UNICOS Configurations

The wall clock time results are shown in Tables 2 and 3. These results are obtained using $T_{wc,1}$ and $T_{wc,2}$, respectively. The error on the mean represents the standard deviation of the *mean* of the distribution. The error bars on Figs. 7 and 8 similarly represent the error on the mean of that quantity. Figures 1 through 5 show that none of the distributions measured in this experiment appear to be normal (Gaussian) distributions. This fact means that no conclusions regarding the probability of a particular mean being correct may be drawn, particularly such conclusions as rely on the assumption that a distribution is normal.

Table 2: Wallclock Time Summary ($T_{wc,1}$)

Time Period	UNICOS Version	No. of CPUs	Elapsed Time (sec.) Mean ± error	Median	Std. Dev.	Number
Prime	5.0	1	143.0 ± 9.0	135.8	42.1	22
		4	131.8 ± 18.4	102.1	77.9	18
		8	179.1 ± 20.4	154.0	88.7	19
Off-Prime	5.0	1	130.3 ± 7.6	123.2	33.8	20
		4	161.2 ± 16.8	146.9	78.8	22
		8	162.8 ± 17.4	146.7	93.9	29
All	5.0	1	137.0 ± 6.0	131.0	38.5	42
		4	148.0 ± 12.5	104.0	78.8	40
		8	169.3 ± 13.2	149.0	91.3	48
Prime	5.1.10	1	144 ± 6	143.	41.8	46
		4	91.1±6.7	88.7	46.3	48
		8	115±11	105	70.9	45
Off-Prime	5.1.10	1	123 ± 29	75.5	265	83
		4	42.6±1.9	38.2	16.3	72
		8	49.2±3.1	38.0	28.0	81
All	5.1.10	1	131±19	87.4	214	129
		4	62.0±3.6	51.9	39.6	120
		8	72.8±5.1	58.9	57.3	126
Prime	6.0/6.0.5	1	228±49	173	164	11
		4	71.8 ± 16.5	47.1	49.6	9
		8	56.1 ± 7.8	40.7	29.1	14
Off-Prime	6.0/6.0.5	1	85.1 ± 4.5	77.3	21.2	22
		4	48.8 ± 5.0	39.8	21.4	18
		8	35.0 ± 2.8	31.6	13.2	23
All	6.0/6.0.5	1	133 ± 20	79.9	116	33
		4	56.5 ± 6.6	43.9	34.3	27
		8	43.0 ± 3.7	33.0	22.8	37
Prime	6.0.1.2	1	180 ± 40	146	121	9
		4	107 ± 42	53.3	110.	7
		8	81.8 ± 18.7	49.0	69.8	14
Off-Prime	6.0.1.2	1	92.4 ± 4.2	89.0	22.0	28
		4	45.4 ± 2.8	44.1	13.8	24
		8	46.3 ± 7.4	31.5	35.7	23
All	6.0.1.2	1	114 ± 12	91.3	71.3	37
		4	59.2 ±10.2	45.8	56.8	31
		8	59.7 ± 8.8	39.2	53.3	37
Prime	6.1.4/6.1.5	1	98.4 ± 7.2	87.9	28.7	16
		4	73.5 ± 8.6	64.8	48.7	32
		8	66.9 ±10.8	43.6	65.8	37
Off-Prime	6.1.4/6.1.5	1	88.9 ± 5.4	78.5	31.3	34
		4	50.6 ±5.5	38.1	45.0	67
		8	50.7 ± 3.8	39.1	33.7	77
All	6.1.4/6.1.5	1	92.0 ± 4.3	80.8	30.6	50
		4	58.0 ± 4.8	41.4	47.2	99
		8	56.0 ± 4.4	40.4	46.9	114

Table 3: Wallclock Time Summary ($T_{wc,2}$)

Time Period	UNICOS Version	No. of CPUs	Elapsed Time (sec.) Mean	Median	Std. Dev.	Number
Prime	5.1.10	1	139 ± 6	138.	41.8	46
		4	84.8±6.7	82.7	46.2	48
		8	110±11	105	70.9	45
Off-Prime	5.1.10	1	118 ± 29	71.0	265	83
		4	37.3±1.9	32.0	16.1	72
		8	43.9±3.0	32.3	27.2	81
All	5.1.10	1	125±19	80.9	214	129
		4	56.3±3.6	44.6	39.3	120
		8	67.7±5.1	54.9	57.2	126
Prime	6.0/6.0.5	1	225±49	173	161	11
		4	67.5 ± 15.3	46.1	46.0	9
		8	51.4 ± 7.0	38.6	26.1	14
Off-Prime	6.0/6.0.5	1	81.6 ± 3.8	76.8	17.7	22
		4	47.5 ± 5.0	38.6	21.3	18
		8	33.8 ± 2.7	30.4	13.0	23
All	6.0/6.0.5	1	129 ± 19	79.6	114	33
		4	54.2 ± 6.2	40.3	32.2	27
		8	40.4 ± 3.4	32.2	20.6	37
Prime	6.0.1.2	1	178 ± 40	146	120	9
		4	91.4 ± 39	52.4	104.	7
		8	65.8 ± 16.4	41.8	61.4	14
Off-Prime	6.0.1.2	1	90.6 ± 4.0	88.1	21.4	28
		4	42.6 ± 2.4	43.1	11.9	24
		8	43.6 ± 7.4	30.2	33.8	23
All	6.0.1.2	1	112 ± 12	88.5	70.7	37
		4	53.6 ±9.4	43.7	52.0	31
		8	52.0 ± 7.7	37.9	46.7	37
Prime	6.1.4/6.1.5	1	95.6 ± 7.1	83.8	28.3	16
		4	58.9 ± 4.8	53.4	27.1	32
		8	49.6 ± 8.4	38.7	38.6	37
Off-Prime	6.1.4/6.1.5	1	86.1 ± 5.2	74.2	30.2	34
		4	41.6 ± 3.8	33.5	31.0	67
		8	41.3 ± 3.0	33.6	26.5	77
All	6.1.4/6.1.5	1	89.1 ± 4.2	77.9	29.7	50
		4	47.2 ± 3.1	38.1	30.8	99
		8	44.0 ± 3.4	34.0	36.4	114

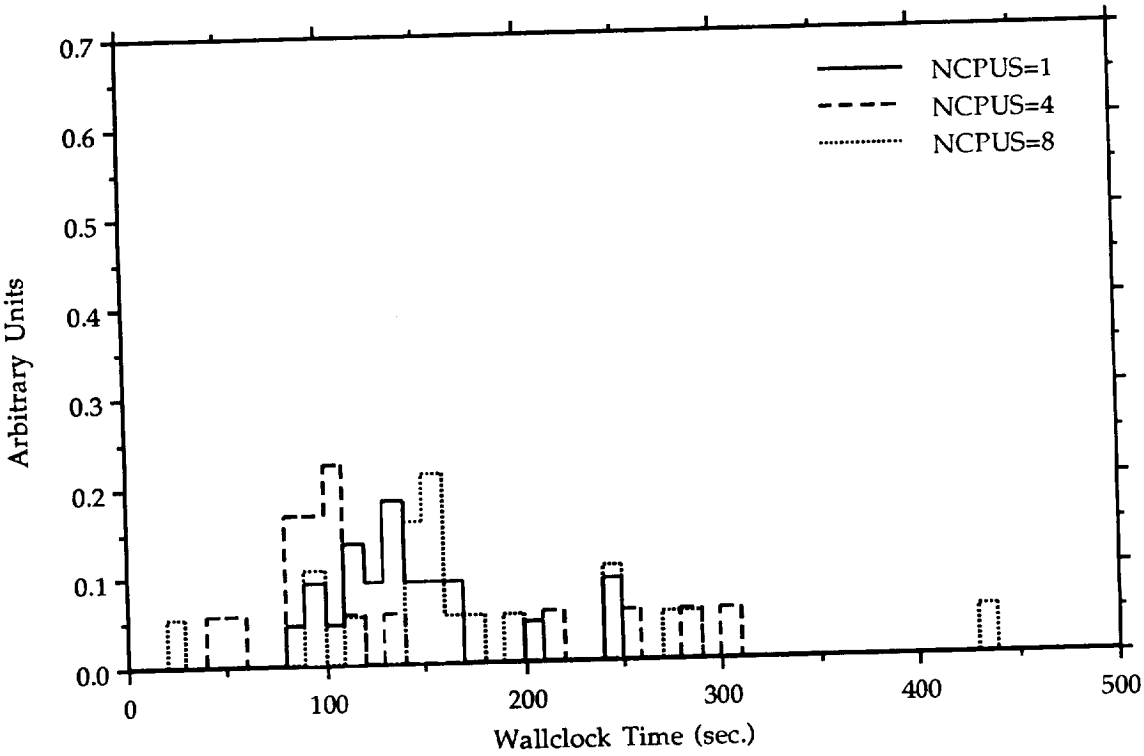

Figure 1a: Prime time, UNICOS 5.0

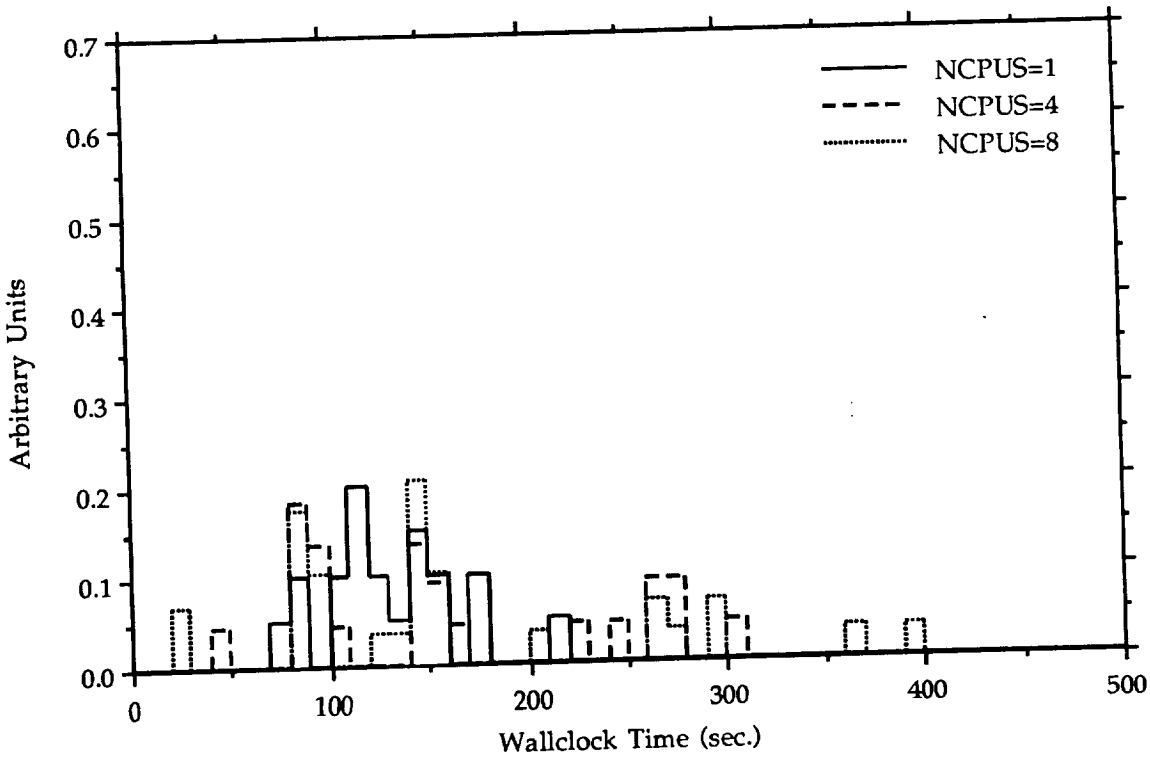

Figure 1b: Off-prime time, UNICOS 5.0

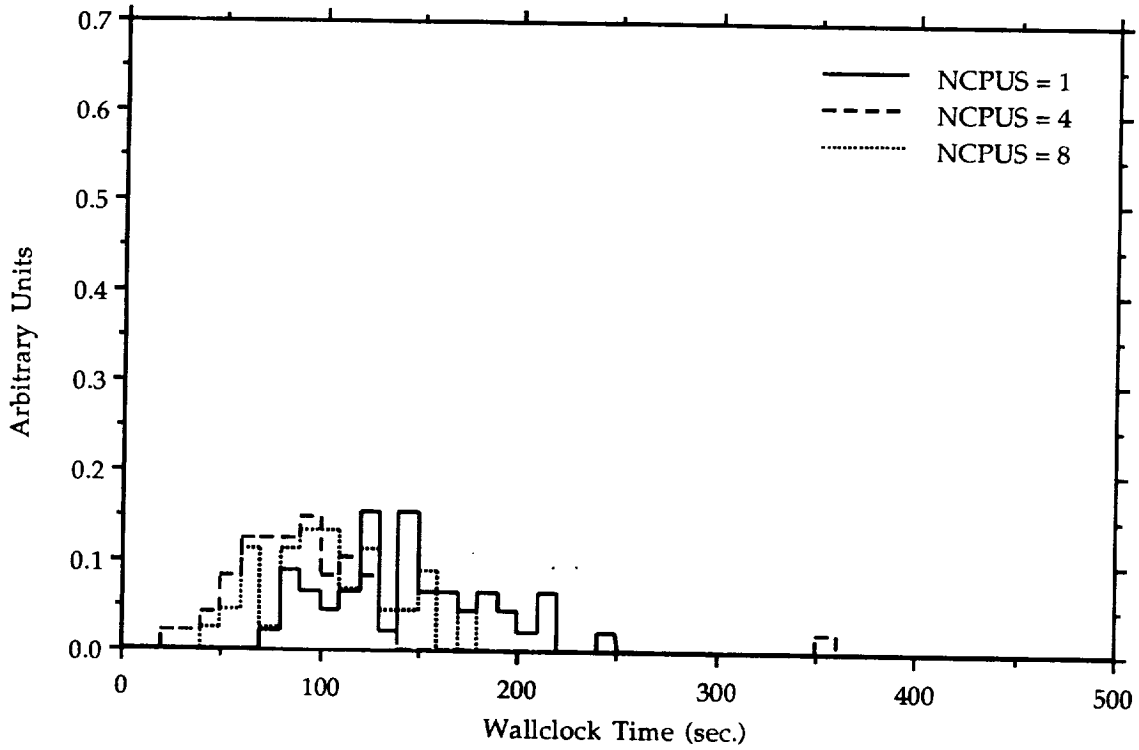

Figure 2a: Prime time, UNICOS 5.1.10

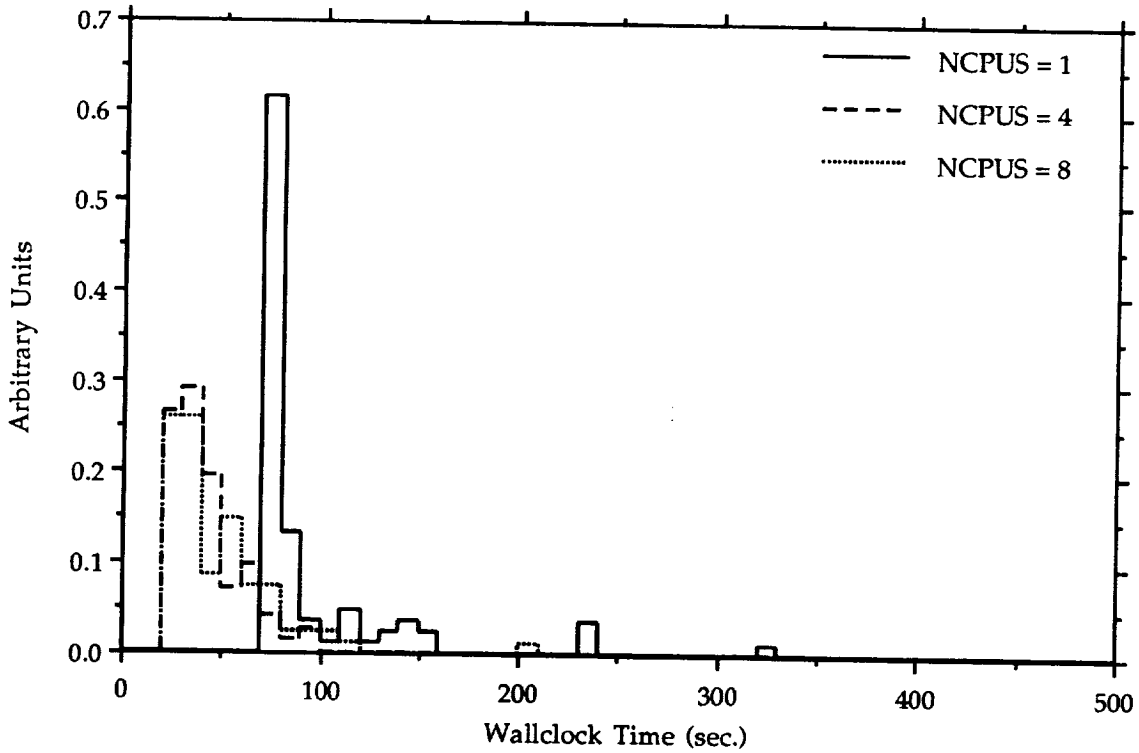

Figure 2b: Off-prime time, UNICOS 5.1.10

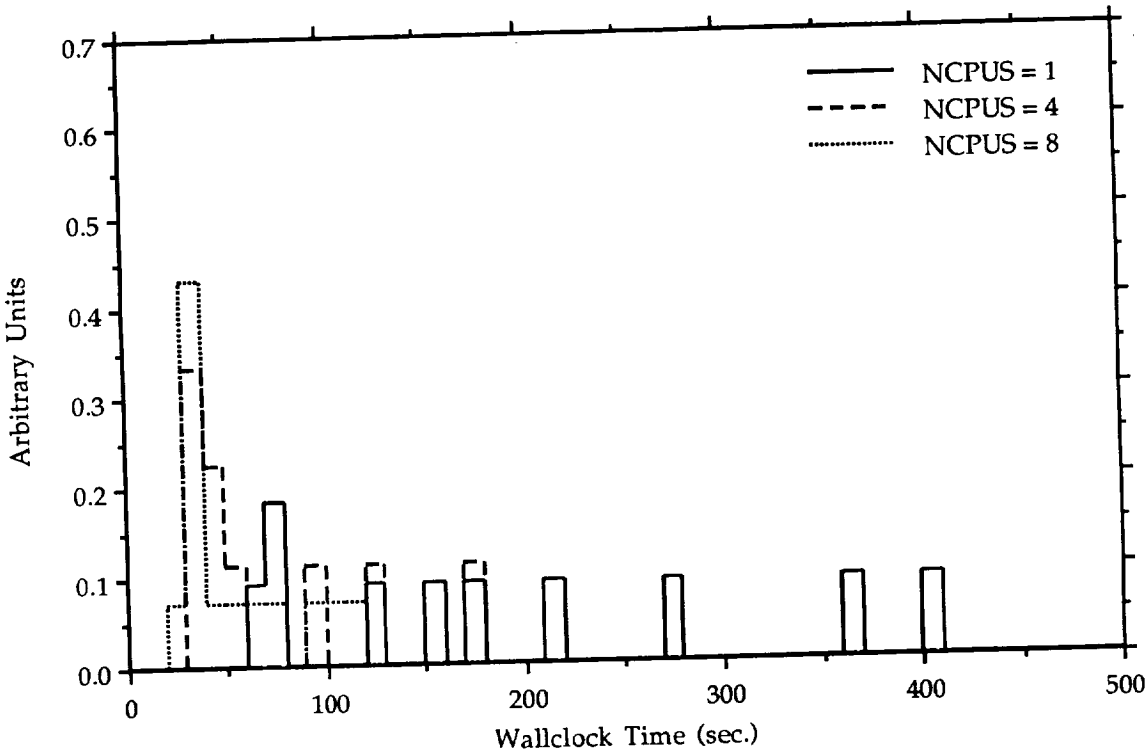

Figure 3a: Prime time, UNICOS 6.0 and 6.0.0.5

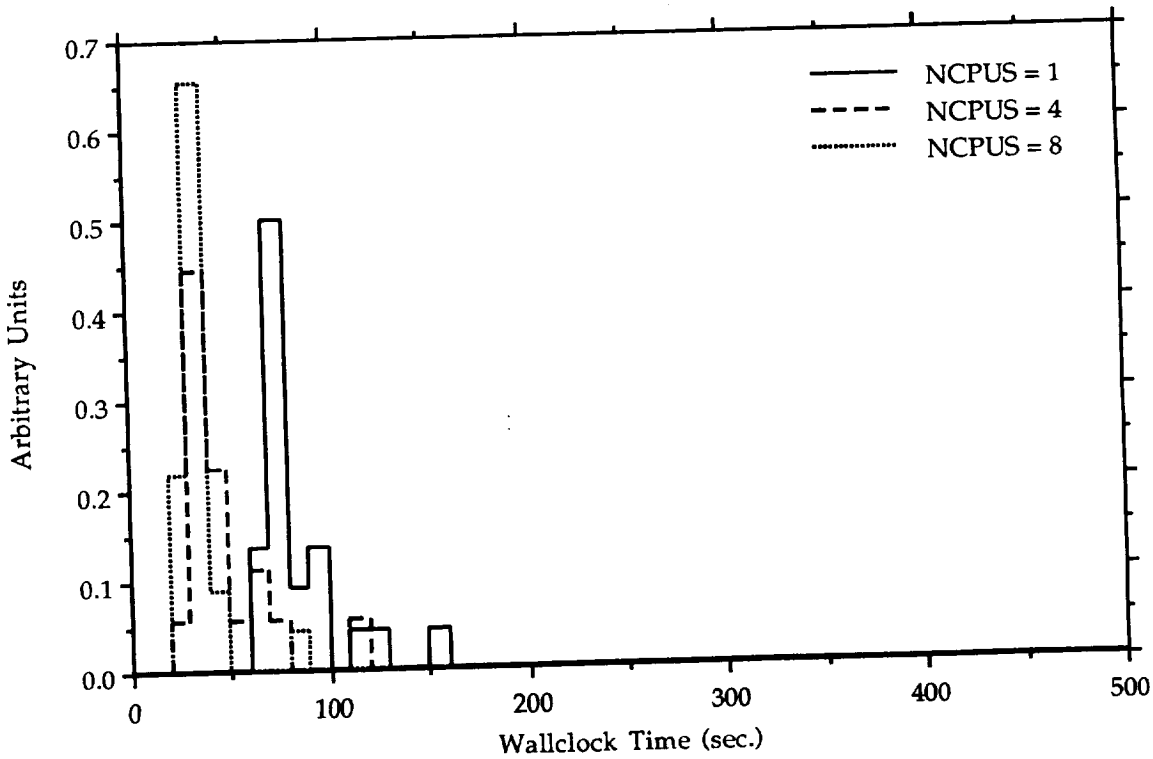

Figure 3b: Off-prime time, UNICOS 6.0 and 6.0.0.5

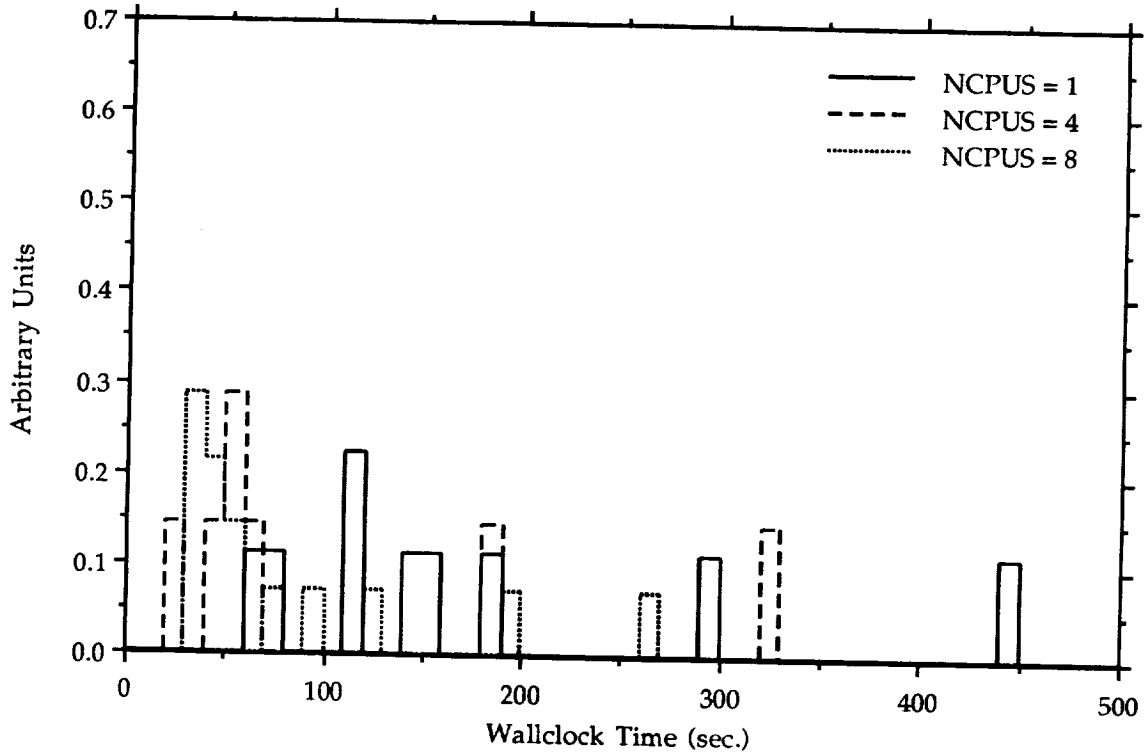

Figure 4a: Prime time, UNICOS 6.0.1.2

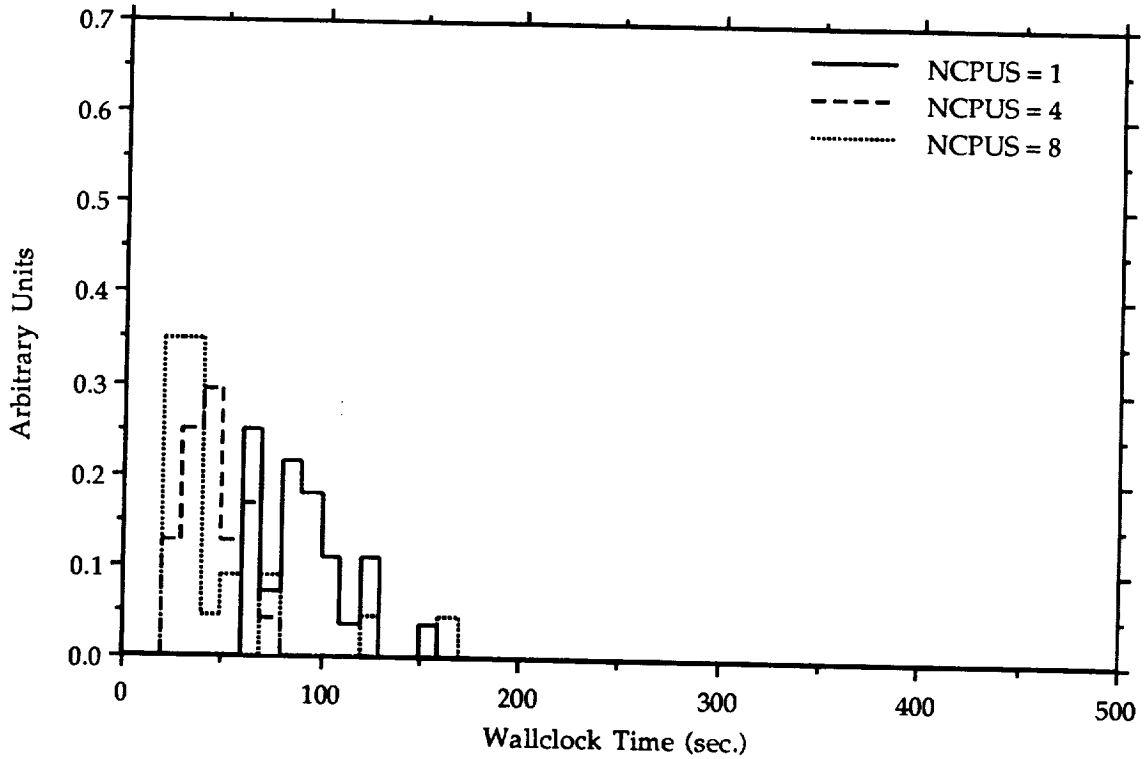

Figure 4b: Off-prime time, UNICOS 6.0.1.2

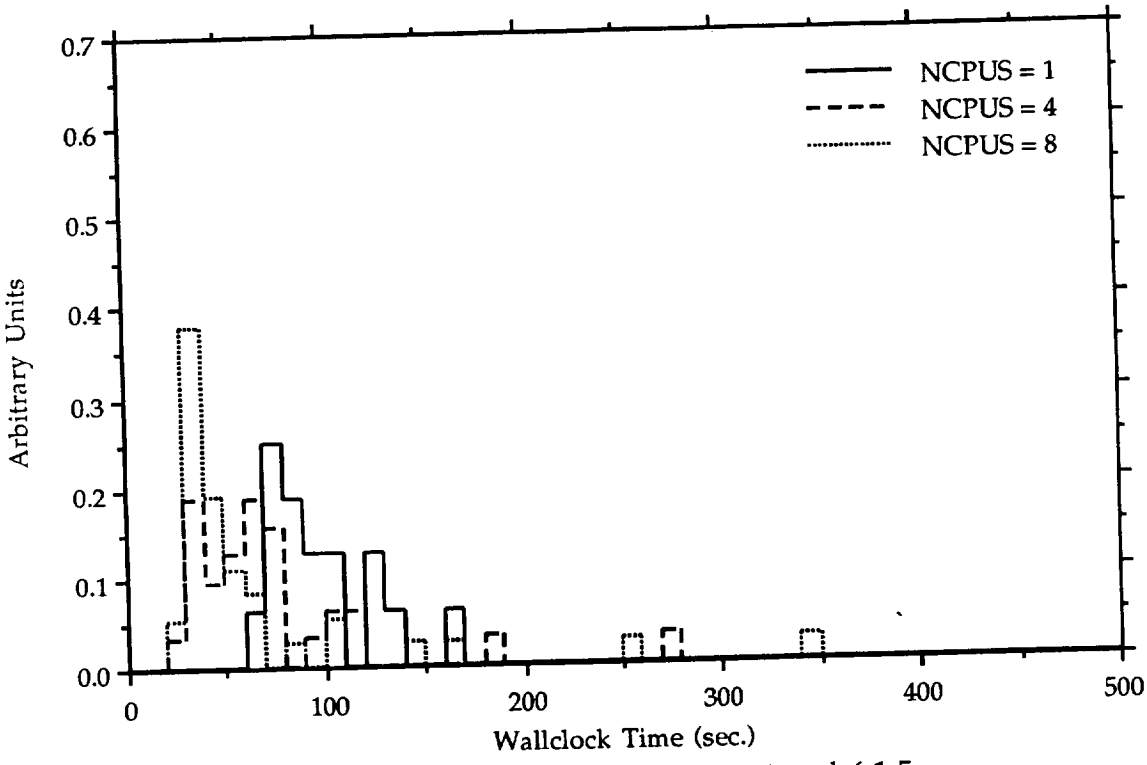

Figure 5a: Prime time, UNICOS 6.1.4 and 6.1.5

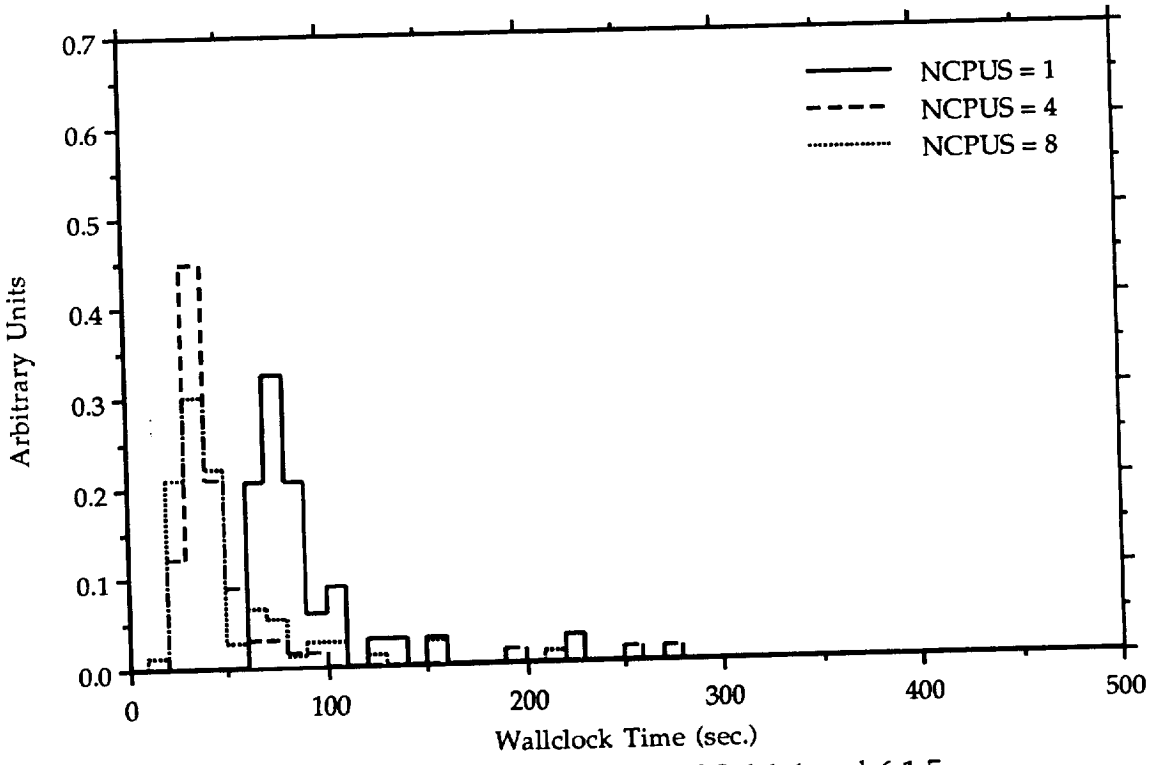

Figure 5b: Off-prime time, UNICOS 6.1.4 and 6.1.5

Figures 1 to 5 show the distribution of wall clock times for each UNICOS configuration, separated into prime and off-prime running periods. A number of distributions (*e.g.*, Fig. 4a) show large non-Gaussian tails at large wall clock times. These runs will be referred to as *outliers*. A second example of non-Gaussian behavior is the cut-off at low elapsed time for each distribution. As the job cannot run faster than it does in a dedicated environment, an asymmetry is introduced into the observed distribution. The area of each distribution is normalized to unity; reading the arbitrary unit off the vertical axis and multiplying it by the histogram bin width thus yields the fraction of runs contained in that bin. The bin width is 10*sec.* in all histograms below.

All data sets, except the 5.0 results, show a clear difference between the distributions of prime and off-prime wallclock times, with smaller mean times in the off-prime execution. There was no benefit to off-prime execution in 5.0. An examination of Table 2 shows that an additional benefit of off-prime execution is reduced fluctuations in the wallclock time as indicated by σ. This effect is most evident for both UNICOS 6.0 datasets. Values of σ obtained under UNICOS 5.0 are indistinguishable between the prime and off-prime execution. This is due to the outliers mentioned above; the outliers increase the skewness of the distribution, limiting the utility of σ as a measure of scatter.

The above results were checked by examining $T_{wc,2}$ (*cf.* Table 3). A comparison of Table 2 and Table 3 shows a significant discrepancy between $T_{wc,1}$ and $T_{wc,2}$ in the UNICOS 6.1 data. A histogram of the difference $\Delta T_{wc} = T_{wc,1} - T_{wc,2}$ is shown in Figure 6. The 6.1 data, and to a much lesser extent the 6.0.12 data, have a number of runs with large discrepancies between the two times. For most of the 6.1 runs, data was available from a script which executes *ps* on SPARK jobs while they are running. These data show that the SPARK processes are swapped out during the time between the printing of the FORTRAN Stop message and the printing of the *ja* Command Report.

The wallclock time discrepancy appears to be an artifact of autotasking. The value of ΔT_{wc} for UNICOS 6.1, 1 CPU runs does not differ significantly from that of earlier datasets. Furthermore, the 4 CPU data under 6.1 has 5 runs where ΔT_{wc} is above 40*sec.*, as opposed to 8 runs above 40*sec.* for the 8 CPU data. We will quote results using $T_{wc,1}$. As the system problem which is keeping these processes swapped out may be solved in future releases of UNICOS, the $T_{wc,2}$ results may be viewed as predictions of improved performance in that eventuality.

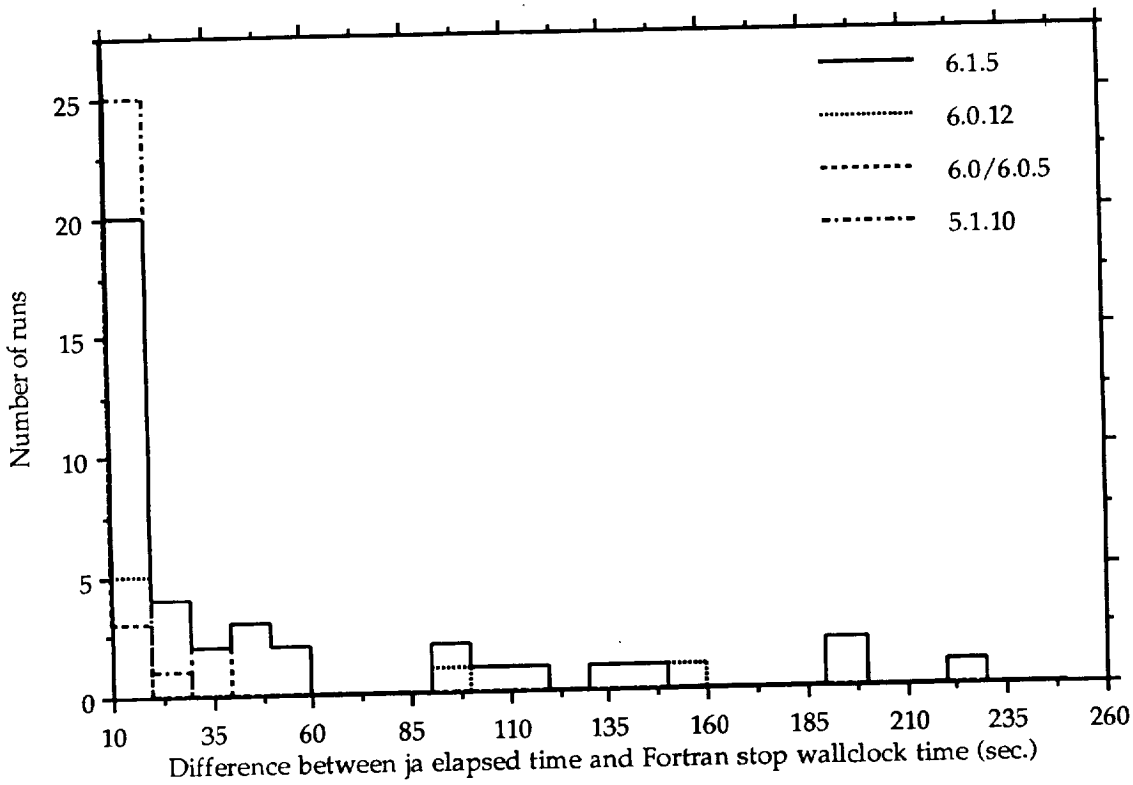

Figure 6: Histogram of ΔT_{wc}

3.2 Comparison of CPU Time Overheads

The autotasking overhead is shown in Table 4 and Fig. 7 below for all datasets. In each case, the overhead increases significantly when NCPUS is increased from 4 to 8. The improvements made in upgrading from UNICOS version 5.1 to 6.0 reduced this penalty by factors of 4 to 2. These numbers suggest that there is little benefit to system throughput when moderately parallelizable codes, such as SPARK, are run with 8 requested CPUs. This conclusion is corroborated by the failure of setting NCPUS to 8 to reduce mean wallclock times in all but the 6.0/6.0.5 dataset. Even in that dataset, the modest reduction in mean wallclock times would not result in a significant improvement in system throughput.

It is also clear that the overhead in the UNICOS 6.0.12 and 6.1.4/6.1.5 datasets is indistinguishable from that of the UNICOS 5.0 data, and much larger than that of UNICOS 6.0/6.0.5. This leads to the conclusion that there are factors present in the system which have as much effect on the overhead as did the major autotasking upgrade put in place with UNICOS 6. Effective use of autotasking for improving system throughput will require the identification of these factors.

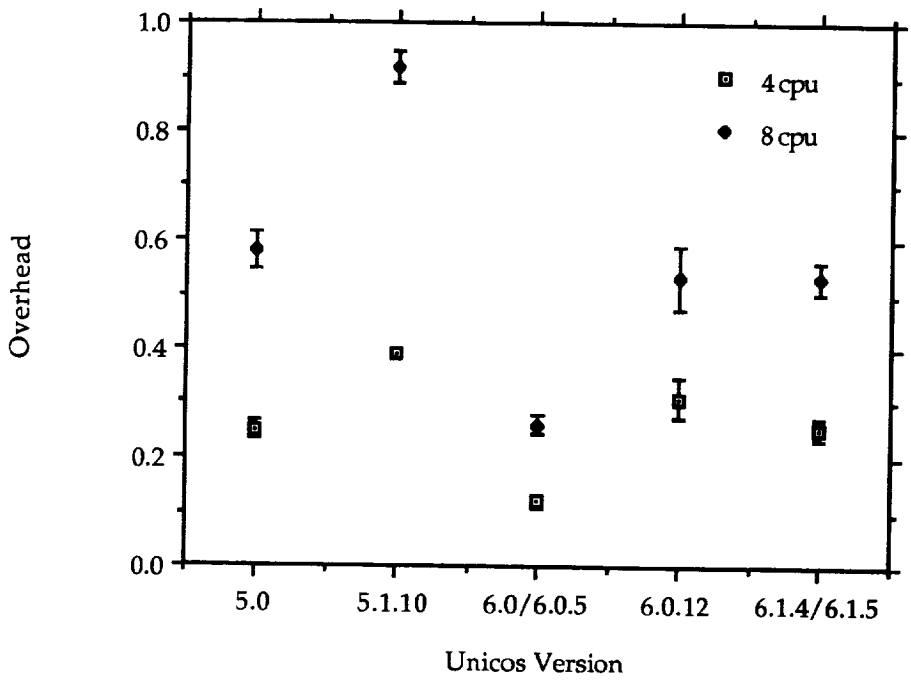

Figure 7: CPU time overhead for each dataset

The data were examined to determine if the overhead increases between the 4 and 8 CPU data were due to system CPU time, user CPU time, or semaphore wait time. For the UNICOS 5 runs, the increase was largely due to the system overhead, with a smaller (less than 20%) contribution from the semaphore wait time and less than 5% from user CPU time. Under UNICOS 6, the increase was due to the user and system CPU times in approximately equal proportions, with a negligible contribution (less than 2%) from the semaphore wait time.

The overhead has also been calculated separately for prime and off-prime running. Decreases in overhead of five to ten percent for off-prime time with respect to prime time running were found in both UNICOS 5 datasets for multi-CPU runs. Running under UNICOS 6, off-prime overheads ranged from 7 percent lower to 2 percent higher for the multi-CPU data. No significant variation was seen between UNICOS 6 or UNICOS 5 datasets with respect to running period. Single CPU runs showed no differences between prime and off-prime overheads. The evidence then suggests that the overhead did not vary dramatically with running period, and that it decreased when UNICOS 6 was installed.

It has been suggested that in future releases of UNICOS the compiler should implement full autotasking by default. The default value of NCPUS (the number of CPUs requested) is set to eight. Given the higher overheads (and resulting charges) with 8 CPUs, and the very similar performance of jobs requesting 4 CPUs compared to jobs requesting 8 CPUs (as observed in Section

3.1), it is questionable that either user or system throughput will be well served by this configuration.

Table 4
CPU Time Overhead

UNICOS Version	No. of CPUs	Overhead
5.0	1	1
	4	0.250 ± .018
	8	0.580 ± .036
5.1.10	1	1
	4	0.392 ± .010
	8	0.917 ± .030
6.0/6.0.5	1	1
	4	0.121 ± .013
	8	0.260 ± .018
6.0.12	1	1
	4	0.309 ± .038
	8	0.532 ± .060
6.1.4/6.1.5	1	1
	4	0.205 ± .021
	8	0.526 ± .042

3.3 Comparison of Average Number of Concurrent CPUs and Observed Speedup

The *ja* utility reports the average number of concurrent CPUs $\langle n_{CPUs} \rangle$ which a job has used. This may naively be viewed as a measure of how effectively a code is running in parallel. A code for which $\langle n_{CPUs} \rangle$ is 4 could be considered "more parallel" than a code with $\langle n_{CPUs} \rangle$ equal to 3. As this simplification overlooks the effects of the CPU time overhead discussed above, it is interesting to compare the observed speedup of the SPARK jobs with their utilization of concurrent CPUs. Figure 8 shows this comparison as a function of the OS version for the 4 CPU and 8 CPU jobs separately.

The data indicate that $\langle n_{CPUs} \rangle$ is a very poor measure of the effective parallelism of the code. With the exception of the UNICOS 6.0/6.0.5 data, no correlation between $\langle n_{CPUs} \rangle$ and the speedup is observed. Except for that case, a correlation which is contrary to the naive expectation is seen, *i.e.*, larger values of $\langle n_{CPUs} \rangle$ correlate to lower speedups.

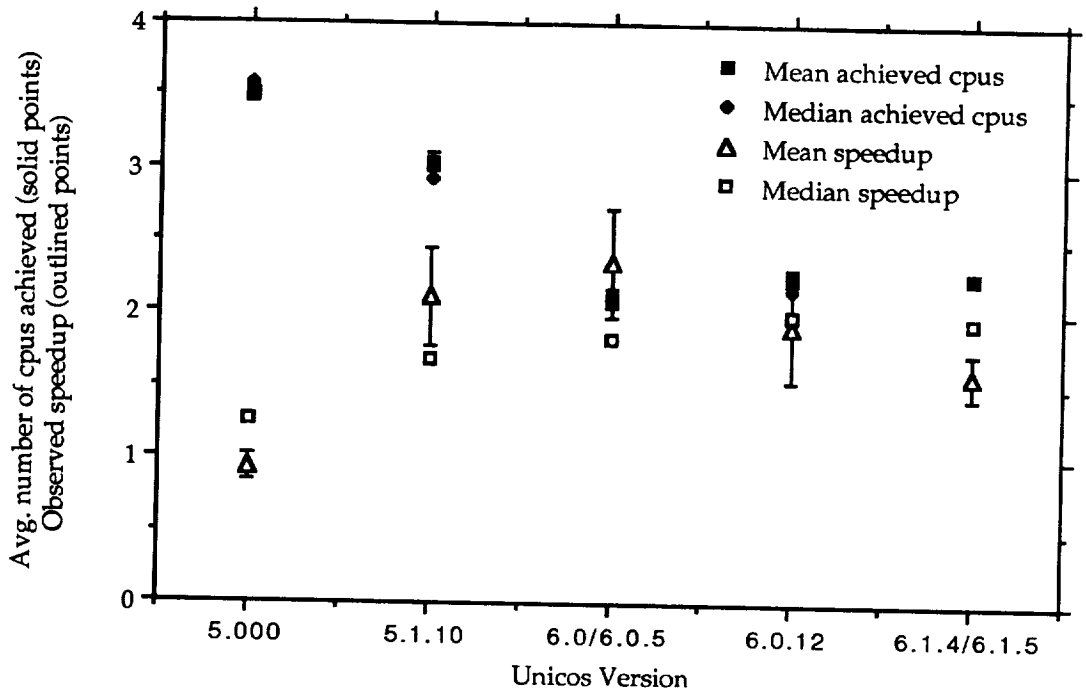

Figure 8a: Results for NCPUS = 4

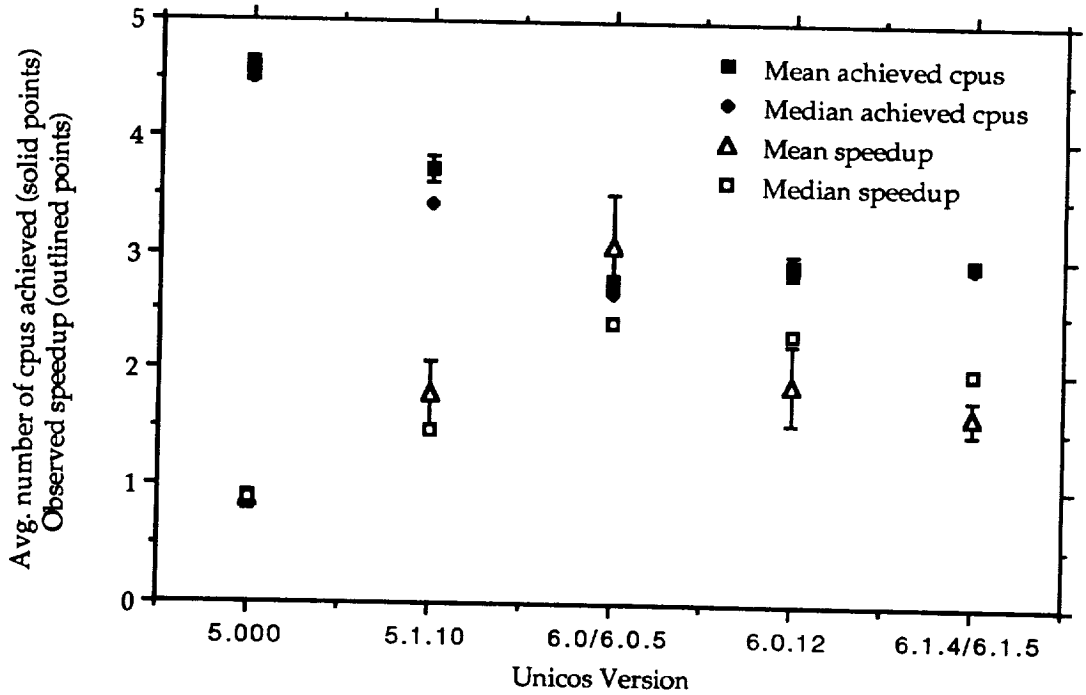

Figure 8b: Results for NCPUS = 8

3.4 Autotasking with 1 CPU vs. a Unitasked Executable

A unitasked version of SPARK was run concurrently with the autotasked executables under UNICOS 6.1. Table 5 shows $<T_{wc,1}>$ and $<T_{CPU}>$ for the unitasked executable and the autotasked executable run with NCPUS set to 1. No significant increase in either time is observed with the autotasked code. There is no penalty to either user or system in running this code autotasked with 1 CPU.

Table 5
Comparison with Unitasked Executable

6.1.4/6.1.5	$<T_{wc,1}>$	$\sigma_{wc,1}$	$<T_{CPU}>$	σ_{CPU}
Unitasked:	92.3 ± 6.3	36.5	68.3 ± 0.3	1.69
1 CPU autotasked:	92.0 ± 4.3	30.6	69.0 ± 0.3	1.76

3.5 Scheduling Parameter Effects

The UNICOS scheduler provides a number of parameters for tuning system performance. As noted above, "outlier" runs appeared under certain OS configurations. A systematic study of the outliers was undertaken in order to determine which, if any, of these parameters might be correlated with these runs. Possible correlations with system load were investigated as part of the same study.

The data for each UNICOS version experiment was examined in the following way. Within an NCPUS=1, 4, or 8 dataset, an outlier was defined to be any run whose wallclock time was greater than $(<T_{wc}> + \sigma_{wc})$. Each dataset was then separated into outlier and "central" runs. All system load variables were compared between these two classes; no strong correlations were found, with the possible exception of the number of interactive logins. This number, determined from *ldave*, averaged about twice as high for outliers as for central runs. It was, however, subject to large fluctuations for both types of runs, rendering firm conclusions difficult.

The scheduling parameters set by *schedv* were then examined for both outlier and central runs. No dependence on any parameter was observed, with the exception of the "hog" parameters. Hogs are processes with memory or CPU time usage over certain values. The value of hog parameters hog_max_mem, memhog, and CPUhog may be examined by using the *schedv* utility. The values were not set permanently, and varied significantly

among the UNICOS version datasets. For all running periods except UNICOS 5.0, the *schedv* bigproc parameter set a memory size (32MW or more on the 128 MW Y-MP) above which no process was swapped out, thus disabling the hog swapping for such processes.

The outliers mainly consist of runs for which the hog_max_mem parameter has a non-zero value (*cf.* Table 6). When this is the case, the UNICOS scheduler will swap out CPU and/or memory hogs if it determines that the overall throughput of the system will benefit.

These data indicate that autotasked jobs would see the best performance if they were run in a queue for which the hog scheduling parameters were ignored. Under the system as it was configured during these measurements, this problem would not have occurred for autotasked jobs requiring more than 16 MW of main memory due to the *schedv* bigproc setting.

Table 6
Effects of Hog Scheduling Parameter

UNICOS Version	Event Class	Number with hog option on	Number with hog option off
5.0	Outlier	24	0
	Central	106	0
5.1.10	Outlier	27	2
	Central	181	165
6.0/6.0.5	Outlier	11	2
	Central	29	56
6.0.1.2	Outlier	8	2
	Central	17	78
6.1.1.4/6.1.1.5	Outlier	15	13
	Central	46	110

3.6 Effect of Running in Different NQS Queues

With the exception of the UNICOS 6.1 data, all SPARK jobs were run in the 10MW/20minute NQS queue (the "normal" queue). The UNICOS 6.1 runs were distributed between that queue and the 32MW/5minutes ("debug") queue. The relevant data are presented below in Table 7. When the wallclock times of runs from the two queues are plotted (*cf.* Fig. 9), the shapes of the distributions differ in the low-time region (20-60*sec.*). This discrepancy may be due to the differing *nice* values of the two queues, which are 12 and 20 for the 32MW/5minutes and 10MW/20minute queues, respectively. (Recall that a lower *nice* value results in higher CPU priority.) There are also more

outliers in the normal queue than in the debug queue. The mean and median wallclock times are in better agreement for the debug queue data, reflecting the relative dearth of outliers.

Table 7: Comparison of debug and normal queue data, UNICOS 6.1			
	NCPUS = 1	NCPUS = 4	NCPUS = 8
$<T_{wc,1}>_{normal}$ - $<T_{wc,1}>_{debug}$	3 ± 9	28 ± 7	14 ± 8
Normal: $\sigma_{wc,1}$	27	53	42
Debug: $\sigma_{wc,1}$	35	12	27

There are suggestive differences between the 4 CPU jobs and the 8 CPU jobs when the data from each queue are compared. The standard deviation of the wallclock time is smaller for the multi-CPU jobs in the debug queue, especially for the 4 CPU jobs. The mean wallclock time $<T_{wc,1}>$ is (6 ± 8) seconds larger for 4 CPU jobs than for 8 CPU jobs in the normal queue, but is (8 ± 6) seconds smaller for the 4 CPU jobs in the debug queue. Given the statistical errors, this is not strong evidence that the 4 CPU jobs take better advantage of the debug queues than do the 8 CPU jobs. As an additional check, the overhead for the 4 and 8 CPU jobs in each queue is shown in Table 8; we see no variation for the 8 CPU jobs, while the 4 CPU jobs' overhead improves significantly by running in the debug queue. While this result is not conclusive, it does show that the possibility of smaller values of NCPUS taking better advantage of the smaller *nice* values of the debug queue cannot be ignored.

Table 8: Overhead for autotasked UNICOS 6.1 jobs		
NCPUS	Debug queue	Normal queue
4	0.16 ± .02	0.29 ± .02
8	0.51 ± .06	0.54 ± .03

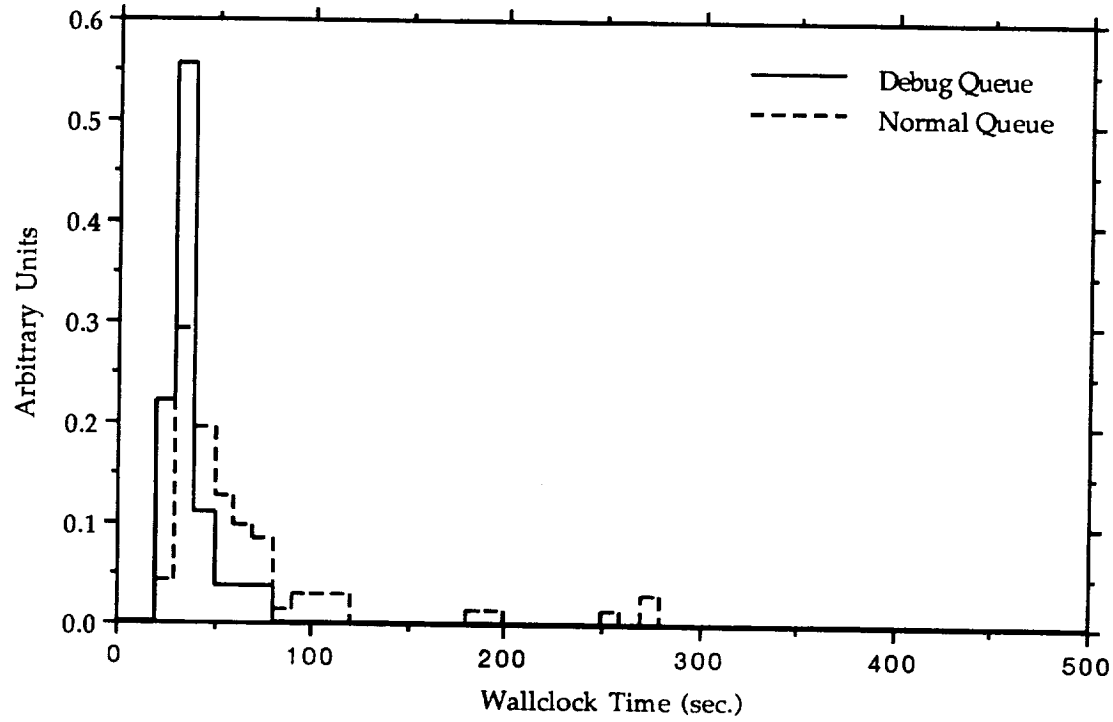

Figure 9a: 4 CPU data, UNICOS 6.1.5

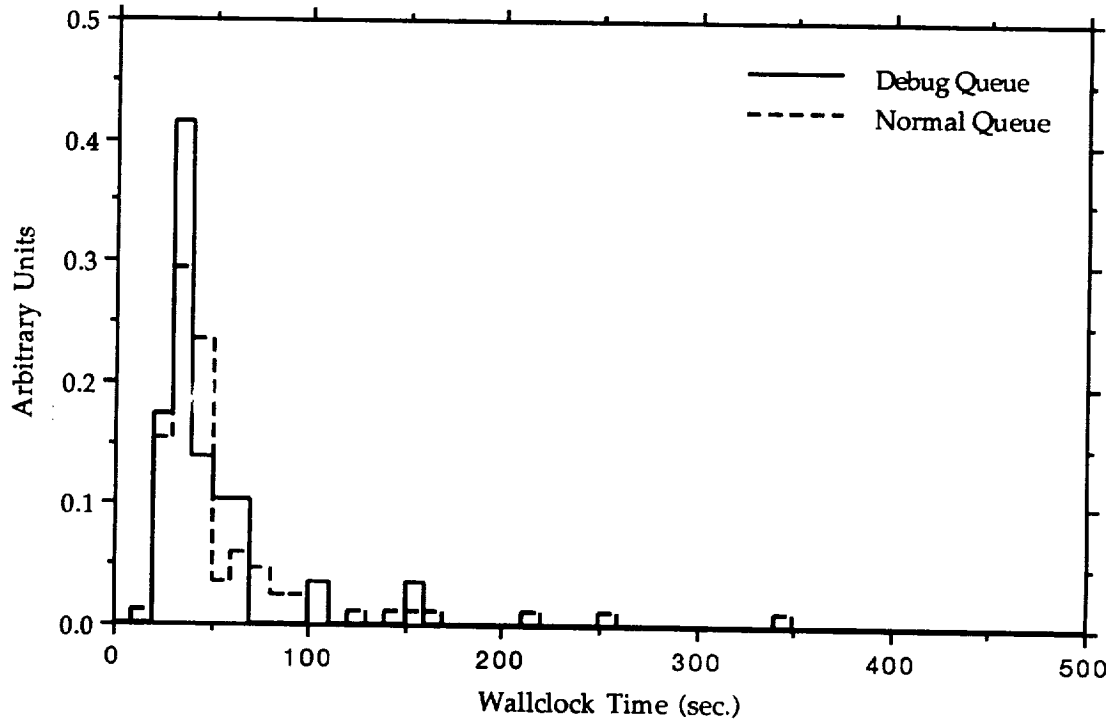

Figure 9b: 8 CPU data, UNICOS 6.1.5

3.7 Study of User Charges and Algorithms

For the period of time covered by these measurements, autotasking and multi-tasking users of the NAS Y-MP were assessed a charge given by the equation:

$$C_\alpha = c_0 * (T_{CPU} - C^* \Sigma (T_{c,i} * c_i)) = c_0 * T_{CPU} - \Sigma (T_{c,i} * q_i)$$

All time variables used here and below (*e.g.* $T_{c,i}$) are defined in Section 1.6 above. The constants c_i are given by Σk, where C is 0.1 and the sum runs over k from 1 to (i-1). The constants c_0 and q_i are shown in Table 9. Users are charged in units of SBUs (system billing units). This algorithm is structured to provide the greatest benefit to codes for which $\langle n_{CPUs} \rangle$ is large.

Table 9
NAS Charge Rate Constants ("old" formula, C_α)

	c_0	q_3	q_3	q_4	q_5	q_6	q_7	q_8
Value (SBU/s):	.03167	.003167	.0095	.03167	.03167	.0475	.0665	.08867

Table 10 shows the mean and standard deviation of the CPU charges which would have been obtained for our runs with various charging algorithms. Using the equation for C_α above, a user running SPARK with NCPUS set to 4 incurs a charge increased by between 10 and 25% above that obtained with 1 CPU. A user running with 8 CPUs suffers larger penalties. This occurs because the increase in CPU time overhead (*cf.* Fig. 7) is larger than the multi-CPU charge reduction. This might discourage a user from autotasking a code, despite the smaller wallclock time (*cf.* Fig. 8).

An improved charging algorithm is thus needed. Two desirable features of such an algorithm would be a rebate (or at least the absence of a penalty) for a user who runs an autotasked code which displays a speedup with only moderate overhead, and a mechanism for penalizing codes with higher overheads. In the present instance, the average charge of the 4 CPU SPARK jobs would be less than that of the 1 CPU SPARK jobs, but the 8 CPU jobs would cost the same as (or more than) the 1 CPU jobs. Note that while this result conflicts with the earlier goal of rewarding users who use many CPUs, it is more beneficial to the system.

One way to effect such an improvement is to modify the value of the charge constants in such a way as to reduce or eliminate the CPU charge penalty. Another approach is to adopt a new algorithm. Care must be taken in designing such an algorithm. For example, using $\langle n_{CPUs} \rangle$ or any quantity directly related to $\langle n_{CPUs} \rangle$ in a charging algorithm would not be suitable (*cf.* Section 3.3).

Coincident with the upgrade to 256 MW of main memory, separate multitasking (mt) queues were established. The charge for these queues is:

$$C_\beta = c_0 * (4T_{ct} + T_{sys})$$

where c_0 is given in Table 9. Thus, NAS charges a parallel job *as if* it uses 4 CPUs. Should a job use more than 4 CPUs, then the CPU time in excess of 4 times the connect time is free. Using less than 4 CPUs results in a larger charge than for a unitasked code. In practice, a code would require $<n_{CPUs}>$ larger than 4 to break even, due to autotasking CPU time overhead. Table 10 shows this charge for the SPARK data. The charge for jobs requesting 1 CPU was calculated using T_{usr} rather than T_{ct}. Since no dataset except UNICOS 5.0 obtained $<n_{CPUs}>$ larger than 4, it is not surprising that no reductions below the normal 1 CPU charge are seen. The 8 CPU jobs do fare slightly better than the 4 CPU jobs. This algorithm succeeds in counter-balancing the 50% overheads present in the 8 CPU data.

Recent data obtained shows that large memory jobs run in the mt queues similarly can obtain relatively high values (up to seven) of $<n_{CPUs}>$, although the speedups observed were less than four [18]. Since fewer than four of the seven CPUs are being used effectively, such jobs consume almost twice as much CPU time as they would using a single CPU. In this case, limiting the number of CPUs which are charged against the job to four reduces but does not eliminate the additional charge resulting from the longer CPU time. Moreover, system overhead for running the job has increased substantially.

Table 10
CPU Charges (SBUs)

UNICOS Version	No. of CPUs	$<C_\alpha>$	σC_α	$<C_\beta>$	σC_β	$<C_\gamma>$	σC_γ
5.0	1	2.31	0.02	9.21	0.08	2.31	0.02
	4	2.31	0.28	3.30	0.60	1.76	0.27
	8	2.67	0.62	3.36	0.83	2.33	0.59
5.1.10	1	2.23	0.03	8.81	0.10	2.23	0.03
	4	2.50	0.25	4.13	0.87	1.64	0.32
	8	3.34	0.74	4.80	1.29	2.46	0.79
6.0/6.0.5	1	2.22	0.02	8.80	0.04	2.22	0.03
	4	2.45	0.14	4.70	0.67	1.83	0.15
	8	2.68	0.22	3.99	0.50	2.00	0.22
6.0.12	1	2.18	0.08	8.55	0.08	2.18	0.08
	4	2.77	0.42	5.02	1.09	2.07	0.39
	8	3.16	0.74	4.60	1.44	2.38	0.66
6.1.4/6.1.5	1	2.18	0.06	8.57	0.10	2.18	0.06
	4	2.66	0.37	4.77	1.04	1.98	0.33
	8	3.16	0.63	4.46	1.01	2.39	0.55

As an example of what results may be obtained by modifying the constants in the original charging scheme, we also tried an algorithm of the form:

$$C_\gamma = c_0 * (T_{CPU} - C * \Sigma (T_{ct,i} * i))$$

This formula attempts to reward autotasking jobs with low overhead, rather than those with high $<n_{CPUs}>$. The constant C was tuned using the UNICOS 6.1 data to give the 4 CPU SPARK jobs a small rebate, while giving the 8 CPU jobs a small penalty. The results, shown in Table 10, used a C equal to 0.35. A comparison of Table 10 and Table 4 shows that the reduction in overhead for the UNICOS 6.0/6.0.5 data is reflected in smaller values for C_γ. In particular, the 8 CPU runs cost *less* than the 1 CPU runs for that dataset. An additional benefit of this algorithm is that the user may predict CPU charges more reliably, which is reflected by the smaller values of $\sigma_{C\gamma}$ as compared to $\sigma_{C\beta}$.

A compromise between the goals of rewarding high $<n_{CPUs}>$ and penalizing high overhead might be achieved by using i^2 rather than i in the formula in the preceding paragraph. A similar analysis to that above shows than a C of 0.10 yields reasonable charges for SPARK. Further study would be required to understand the balance between the two goals should this type of scheme be implemented on supercomputers with more processors such as the 16 processor CRI Y-MP C-90.

4 Summary

Autotasking overhead over the course of this experiment did not significantly change, except for UNICOS 5.1 where it was significantly worse. UNICOS version 6.0 corrected the overhead and wall clock turnaround problems of the previous versions, but subsequent UNICOS upgrades display increasing overheads to the extent that the average autotasking overhead has returned to the high level observed in UNICOS 5 releases. The observed overheads were significantly larger for the set of jobs that requested 8 CPUs than for the set of jobs that requested 4 CPUs, for all UNICOS versions.

On the other hand, with the upgrade to UNICOS 6.0, wall clock turnarounds are excellent for the tested code, which is a necessity in the NAS environment. Speedups for UNICOS 6 releases show consistent wall clock speedups in the workload of around 2, which is quite good. The observed speedups were very similar for the set of jobs that requested 8 CPUs and the set that requested 4 CPUs. Given the higher overheads incurred by jobs which request 8 CPUs, the planned default compiler option of autotasking turned on is ill-advised.

We observed a problem with jobs swapping out after the FORTRAN STOP statement execution in UNICOS versions 6.0.12 and 6.1. Fixing this problem could improve the observed speedups by up to 25%. In addition, the particular job tested does not achieve its maximum speedup in the workload

because the operating system is sensibly biased via the hog scheduling parameters toward maximizing the CPU time of the largest memory jobs. There is also weak evidence for an observable correlation between wall clock speedup and the UNICOS *nice* value.

The original NAS algorithm for determining charges to the user discourages autotasking in the workload. The new NAS algorithm to be applied to jobs run in the NQS multitasking queues also discourages NAS users from using autotasking on the basis of the charges they receive. This algorithm also favors jobs requesting 8 CPUs over those that request less, although the jobs requesting 8 CPUs experienced significantly higher overhead and presumably degraded system throughput.

We present a charging algorithm that has the following desirable characteristics when applied to the data: higher overhead jobs requesting 8 CPUs are penalized when compared to moderate overhead jobs requesting 4 CPUs, thereby providing a charging incentive to NAS users to use autotasking in an manner that provides them with significantly improved turnaround while also maintaining system throughput.

5 Suggestions for Improvements

An improvement to the methodology of this experiment (to improve the accuracy of the conclusions) would be to use several codes, all with significantly longer execution time and larger memory requirements. This would enable more accurate observation of the effects of hog scheduling parameters, and better imitate the NAS workload. Useful information needs to be obtained on the effects of differing amounts of parallelism and I/O. Sampling a larger number of requested CPUs, i.e., sampling NCPUS=1,2,4,6,8, would enable a better understanding of the relationship of CPUs requested to overhead and speedup. This would also provide further tests to determine correct *nice* values for queues.

6 Acknowledgements

The authors wish to thank Toby Harness and David McNab of NAS RNS, for patiently elucidating the intricacies of the NAS charging algorithms. They wish also to thank Duane Carbon and Robert Bergeron of NAS RND, and Howard Walter of NAS RNS for their constructive remarks.

References

[1] Cray Research, Inc., *CF77 Compiling System, Volume 4: Parallel Processing Guide*, Pub. No. SG-3074 5.0, Cray Research, Inc., 1991, pp. 201-234.

[2] R.A. Fatoohi, "Multitasking on the Cray-2 and Cray Y-MP: An Experimental Study," Report RNR-88-001, Dec. 1988, RNR Branch, NAS Systems Division, NASA Ames Research Center, MS T045-1, Moffett Field, CA, 94035-1000.

[3] Rod Fatoohi, and Seokkwan Yoon, "Multitasking the INS3D-LU Code on the Cray Y-MP," Report RNR 91-015, April, 1991, RNR Branch, NAS Systems Division, NASA Ames Research Center, MS T045-1, Moffett Field, CA, 94035-1000.

[4] Russell Carter, "Autotasked Performance of a NASA CFD Code," Report RND 90-003, December, 1990, RND Branch, NAS Systems Division, NASA Ames Research Center, MS 258-5, Moffett Field, CA, 94035-1000.

[5] Charles Grassl, "Benchmarking Autotasking," *Cray Channels*, Summer, 1989, pp. 14-16.

[6] Michael Bieterman, "The Impact of Microtasked Applications in a Multiprogrammed Environment," *Cray Channels*, Summer, 1989.

[7] Robert J. Bergeron, "Parallel Processing in a Computationally Intensive Workload," Report RND-91-007, October, 1991, RND Branch, NAS Systems Division, NASA Ames Research Center, MS 258-5, Moffett Field, CA, 94035-1000.

[8] Markus A. Linn, Wolfgang E. Nagel, Marga Vaessen, and Ulrich Detert, "UNICOS 5.1 vs. UNICOS 6.0: Performance Evaluation of Autotasking," in *Proceedings of the Fall 1991 Cray Users Group Conference*, pp. 321-330.

[9] Lucien F. Kramer, "Batch Autotasking Efficiency On UNICOS 6.0," in *Proceedings of the Spring 1991 Cray Users Group Conference*, p. 37.

[10] Koushik K. Ghosh, "Performance of Multitasked Job-mixes on a Cray Y-MP8 under UNICOS 6.0," in *Proceedings of the Spring 1991 Cray Users Group Conference*, pp. 41-56.

[11] Frank Barriuso, Jim Kohn, Suzanne LaCroix, and Steve Reinhardt, "Improvements to Nondedicated Performance of Autotasking Programs on Cray Y-MP Computer Systems," in *Proceedings of the Fall 1990 Cray Users Group Conference*, pp. 295-300.

[12] Wayne Pfeiffer and John Kim, "Autotasking Performance on the Cray Y-MP," in *Proceedings of the Fall 1991 Cray Users Group Conference*, pp. 292-303.

[13] "NQS-Network Queuing System Version 2", COSMIC, University of Georgia, ARC13179, 1991.

[14] Robert J. Bergeron, "Performance Analysis of the NAS Y-MP Workload," Report RND-90-009, December, 1990, RND Branch, NAS Systems Division, NASA Ames Research Center, MS 258-5, Moffett Field, CA, 94035-1000.

[15] Cray Research, Inc., *UNICOS Performance Utilities Reference Manual*, Pub. No. SR-2040 B, 1989.

[16] J.Philip Drummond, R.Clayton Rogers, and M. Yousuff Hussaini, "A Numerical Model For Supersonic Reacting Mixing Layers," *Computer Methods in Applied Mechanics and Engineering*, 64 1987.

[17] Robert J. Bergeron, "A Performance History of Several Multitasking Codes on the NAS Y-MP," Report RND-91-008, July, 1991, RND Branch, NAS Systems Division, NASA Ames Research Center, MS 258-5, Moffett Field, CA, 94035-1000.

[18] Robert J. Bergeron, Private Communication, May 29, 1992.